THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
000(00) 1–12

DOI: 10.1177/1094342010369118
hpc.sagepub.com
SAGE

Understanding Checkpointing Overheads on Massive-Scale Systems: Analysis of the IBM Blue Gene/P System

Rinku Gupta[1], Harish Naik[1] and Pete Beckman[1]

Abstract

Providing fault tolerance in high-end petascale systems, consisting of millions of hardware components and complex software stacks, is becoming an increasingly challenging task. Checkpointing continues to be the most prevalent technique for providing fault tolerance in such high-end systems. Considerable research has focussed on optimizing checkpointing; however, in practice, checkpointing still involves a high-cost overhead for users. In this paper, we study the checkpointing overhead seen by various applications running on leadership-class machines like the IBM Blue Gene/P at Argonne National Laboratory. In addition to studying popular applications, we design a methodology to help users understand and intelligently choose an optimal checkpointing frequency to reduce the overall checkpointing overhead incurred. In particular, we study the Grid-Based Projector-Augmented Wave application, the Carr-Parrinello Molecular Dynamics application, the Nek5000 computational fluid dynamics application and the Parallel Ocean Program application—and analyze their memory usage and possible checkpointing trends on 65,536 processors of the Blue Gene/P system.

Keywords

Checkpointing, Blue Gene, Fault Tolerance, I/O, Massive scale systems

1 Introduction

The past two decades have seen tremendous growth in the scale, complexity, functionality, and usage of high-end computing (HEC) machines. The Top500 (Dongarra et al.) list shows that performance offered by high-end systems has increased by over eight times in the past five years. Current petascale machines consist of millions of hardware components and complex software stacks and future exascale systems will exponentially increase this complexity. This increasing system complexity has resulted in justified concerns about reliability and fault tolerance in these machines.

High performance computing end-users and applications developers continue to use checkpointing/restart (Koo and Toueg, 1986) as a preferred technique for achieving fault tolerance in their software. Checkpointing requires saving the local state of a process at a specific time and then rolling back (also called recovery/restart) to the latest saved state in the event of a crash during the execution lifetime of a process.

New petascale machines, such as the IBM Blue Gene series (IBM Blue Gene Team, 2008), which are capable of scaling to over several hundreds of thousands of cores, can offer enormous computational power. However, these machines (as well as future exascale machines) utilize a

[1]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, {rgupta, hnaik, beckman}@mcs.anl.gov, April 19, 2010

This paper is an extended version of the paper presented at the International Workshop on Parallel Programming Models and Systems Software for High-End Computing Systems (P2S2), 2009, titled \Analyzing Checkpointing Trends for Applications on the IBM Blue Gene/P System." In this section, we highlight some of the additions done in this version of the paper, as compared to the one presented at the workshop (several other minor changes and explanation details I added in the paper are skipped here): Additional Applications: Together with the diÛerent applications demon- strated in the workshop version of the paper, this version also includes a new computational¯uid dynamics application, the Parallel Ocean Program (POP), which is a popular application for ocean modeling. POP is an integral part of the SPEC 2007 benchmark suite, and represents a wide-class of applications. Additional Experimentation and Analysis: This paper also presents sev-eral additional results, including experiments on larger system sizes and more analysis of the time taken for checkpointing, evaluation of checkpointing param- eters, and other trends. Theoretical Modeling: The paper also presents a new simplied optimal checkpointing model and uses this model to evaluate checkpointing parameters, and other trends. Theoretical Modeling: The paper also presents a new simplied optimal checkpointing model and uses this model to evaluate checkpointing parameters, and other trends. Theoretical Modeling: The paper also presents a new simplied optimal checkpointing model and uses this model to evaluate checkpointing parameters for all the diÛerent applications.

large degree of shared hardware, including shared memory, shared caches, and a shared network infrastructure. Storage on such machines is provided, also, through a shared I/O infrastructure. Large-scale applications, using these machines, use checkpointing for proactive fault tolerance. Saving states of thousands of processes during a checkpoint can result in a heavy demand of I/O and network resources. Since these machines have limited I/O and network resources, frequent checkpointing (and saving these checkpoints to back-end storage), at this scale, can result in longer execution times, especially if the I/O or network resources become a bottleneck. This undermines the end-users' ability to use the machine effectively and may result in wastage of the limited processing cycles allocated to them on such large machines. For applications and users to use these machines in the most efficient manner, we must understand the feasibility of checkpointing and memory trends of these applications on large-scale machines. Information about memory trends of these applications will allow end-users to estimate the checkpointing time and thus make conscious decisions on the checkpointing frequency. While measurement and analysis of memory trends have received some study, investigations have been limited to small-scale systems of up to 64 processors (Sancho et al., 2004). The challenges for checkpointing on large leadership machines are completely different and on a different scale.

In this paper, we study similar checkpointing and memory trends for applications on the IBM Blue Gene/P system (BG/P) at Argonne National Laboratory (ANL). Specifically, we present memory trends of four popular applications: the Grid-Based Projector-Augmented Wave application (GPAW), the Carr-Parrinello Molecular Dynamics application (CPMD), the Nek5000 computational fluid dynamics application, and the Parallel Ocean Program (POP). We also present an analytical model for efficiently computing the optimum checkpoint frequencies and intervals, and we determine the limitations of checkpointing such applications on large systems. Our study and analysis are based on a ''full checkpointing'' technique, in which the entire program state of the processes is stored during the checkpoint operation. This technique is used by the IBM checkpointing library (Sosa and Knudson, 2007) and is the only checkpointing software currently available for the IBM Blue Gene/P machines.

The rest of the paper is organized as follows. Section 2 discusses the background and related work performed in the area of checkpointing. Section 3 provides an overview of the IBM Blue Gene/P system and its checkpointing software. Section 4 presents the applications used in this study. Section 5 discusses our checkpointing model, experiments performed and analysis of the results. In Section 6, we present the conclusions and outline future work.

2 Overview of Checkpointing

Checkpointing methods and optimization techniques have been studied and summarized by several researchers (Plank, 1997; Silva and Silva, 1998; Zomaya, 1995). In this section, we briefly discuss some checkpointing concepts and techniques common to distributed computing environments. We will also discuss work done in the area of checkpointing in (MPI) programs (Schulz et al., 2004) on large scale high-performance computing (HPC) systems. In distributed systems, checkpointing can typically occur at the operating system level or the application level.

Operating system-level checkpointing: Operating system (OS)-level checkpointing is a user-transparent way of implementing checkpointing. In this approach, the user typically needs to specify only the checkpointing interval, with no additional programming effort; other details such as checkpoint contents are handled by the operating system (Barigazzi and Strigini, 1983). OS-level checkpointing for an application involves saving the entire state of the application, inclusive of all processes and temporary data, at the checkpoint time. Since this type of checkpointing does not take into account the internal characteristics and semantics of the application, the total size of the checkpointing data can dramatically increase with system size. On petascale systems, which are I/O bound, this can cause a heavy overhead on the runtime of the application.

Application-level checkpointing: Application-level checkpointing (Zomaya, 1995), also called user-defined checkpointing, is a checkpointing method that enables the user to intelligently decide the placement and contents of the checkpoints. The primary advantage of this approach is that as users semantically understand the nuances of the applications they can place checkpoints, using libraries and preprocessors, at critical areas, potentially decreasing the size of the checkpoint contents and checkpointing time. While this approach requires more programmer effort, it is more portable, since checkpoints can be saved in a machine-independent format, and thus offers better performance and flexibility as compared to the OS-level approach.

An important difference between the two types of checkpointing is that an OS-level checkpoint can be performed at any point during the execution of the application. But application-level checkpoints can only be performed at specified points in the program during the execution (Schulz et al., 2004).

Compiler-level checkpointing (Li and Fuchs, 1990) also exists. In this case, the compiler selects optimal checkpointing locations to minimize checkpointing content. Hybrid techniques, such as compiler-assisted checkpointing coupled with application checkpointing, also have been studied. These provide a certain degree of transparency to the user.

Checkpointing in distributed systems requires that global consistency be maintained across the entire system and the avoidance of the domino effect. One way to achieve this consistency is through *coordinated checkpointing* (Guohong and Singhal, 1998). In coordinated checkpointing, once the decision to checkpoint is made, the program does not progress unless all the checkpoints of all the processes are saved. Coordinated checkpointing requires that system or process components communicate with each other to establish checkpoint start and end times in single or multiple phases. Recovery in this technique is achieved by rolling back all processes to the latest state. Another method to achieve global consistency is through

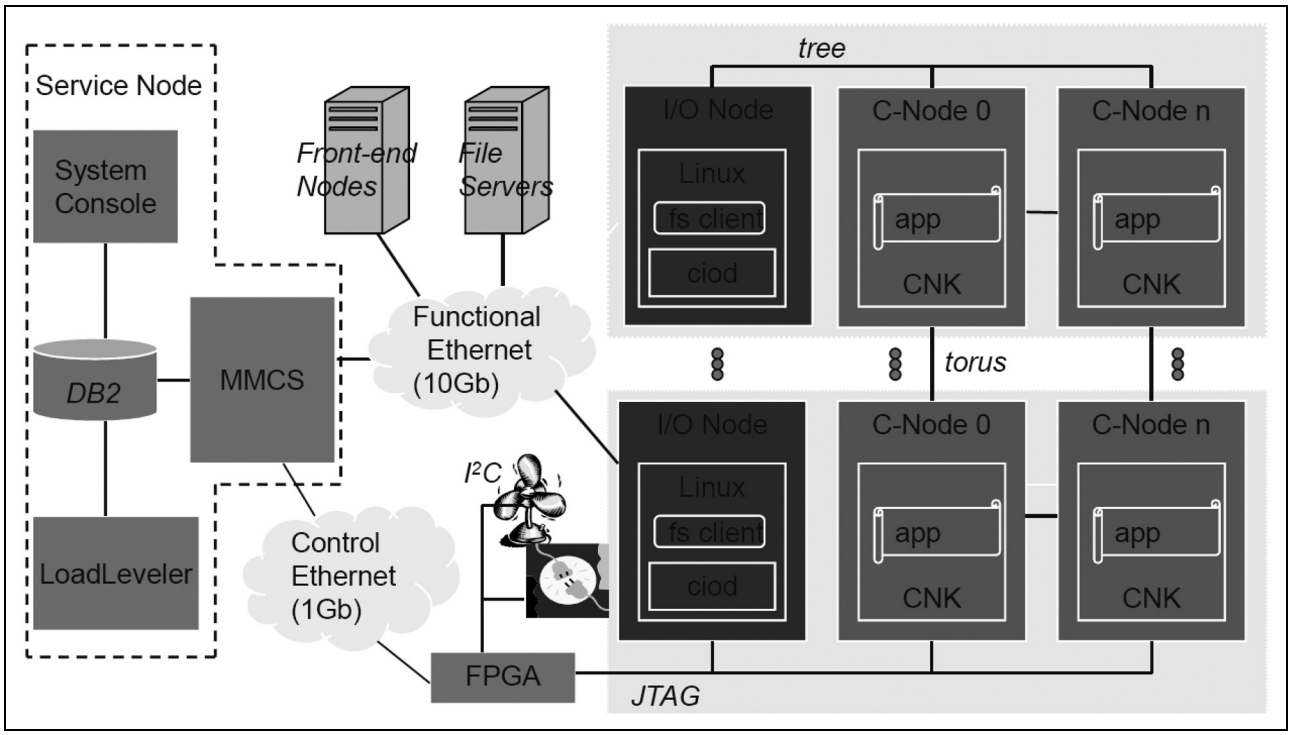

Figure 1. ANL BG/P system architecture. (This figure was taken from the "IBM System Blue Gene/P Solution: Blue Gene/P Application Development" redbook (Sosa and Knudson, 2007).)

independent checkpointing (Zomaya, 1995), which uses synchronous and asynchronous logging of interprocess messages along with independent process checkpointing. In this method, recovery is achieved by rolling back to the faulty process and replaying the messages received by the faulty process. Other techniques such as *adaptive independent checkpointing* (Zomaya, 1995) are based on hybrid coordinated and independent checkpointing techniques.

Checkpointing optimizations have received considerable attention. Two primary techniques that have emerged are *full-memory checkpointing* and *incremental-memory checkpointing* (Plank et al., 1995). In full-memory checkpointing, during each checkpoint instance the entire memory context for that process is saved. In incremental-memory checkpointing, pages that have been modified since the last checkpoint are marked as dirty pages and are saved. Incremental-memory checkpointing, thus, can reduce the amount of memory context that needs to be saved, especially for large systems.

Performing checkpointing operations on large scale systems becomes very challenging in the context of large scale distributed memory systems. A study by Liu et al. (2008) proposes a checkpoint and recovery model for large scale HPC systems. There have also been studies on checkpointing and recovery mechanisms applied to a Network of Workstations (NOW) (Dongsheng et al.). But in a system like Blue Gene/P with 163,840 cores available for computation, the challenges are far more severe.

3 Overview of the Blue Gene/P System

In this section, we discuss the architecture of the IBM Blue Gene/P (IBM Blue Gene Team, 2008) system at Argonne National Laboratory named *Intrepid*. We also provide an overview of the IBM Blue Gene/P checkpointing library.

3.1 The BG/P Hardware Architecture

The *Intrepid* (Desai et al., 2008) system at Argonne National Laboratory is IBM's massively parallel supercomputer consisting of 40 racks of 40,960 quad-core compute nodes (totaling 163,840 processors), 80 TB of memory and a system peak performance of 556 TF. Each compute node consists of four PowerPC 450 processors, operates at 0.85 GHz, and has 2 GB memory. Compute nodes run a lightweight Compute Node Kernel (CNK), which serves as the base operating system. In addition to the compute nodes, 640 input/output (I/O) nodes are used to communicate with the file system. The I/O nodes physically are the same as compute nodes but differ from them in function. The Argonne system is configured to have a 1:64 ratio with a single I/O node managing 64 compute nodes. In addition to the compute and I/O nodes, there exist service and login nodes that allow diagnostics, compilation, scheduling, interactive activities, and administrative tasks to be carried out. This system architecture is depicted in Figure 1[1].

The Intrepid system uses five different networks. A three-dimensional torus network, based on adaptive cut-through routing, interconnects the compute nodes and carries the bulk of the communication while providing low-latency, high bandwidth point-to-point messaging. A collective network interconnects all the compute nodes and I/O nodes. This network is used for broadcasting data and forwarding file-system traffic to I/O nodes. An independent, tree-based, latency-optimized barrier network also

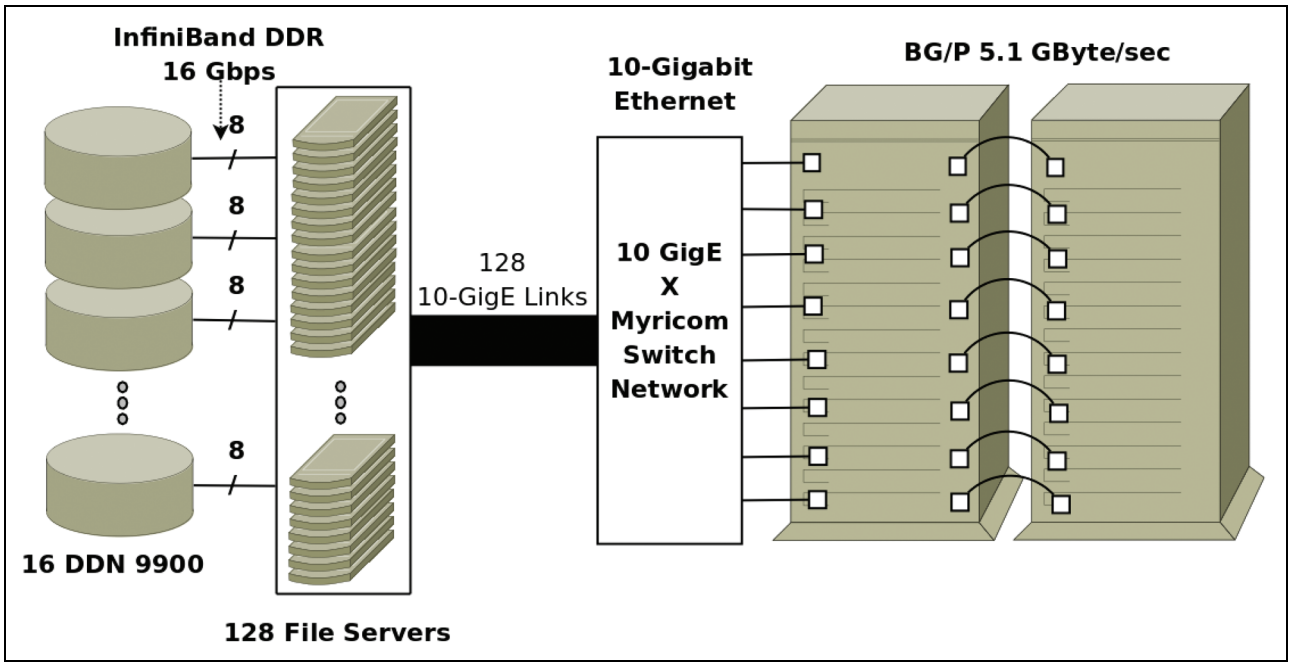

Figure 2. ANL BG/P file system.

exists for fast barrier collective operations. In addition, a dedicated Gigabit Ethernet and JTAG network that connects the I/O nodes and compute nodes to the service nodes is used for diagnostics, debugging, and monitoring. Lastly, a 10-Gigabit Ethernet network, consisting of Myricom switching gear connected in a nonblocking configuration, provides connectivity between the I/O nodes, file servers, and several other storage resources.

As seen in Figure 2, the Intrepid back-end file system architecture includes 16 DataDirect 9900 SAN storage arrays (each with 480 1 TB disks) offering around 8 petabytes of raw storage. Each array is directly connected to eight file servers through eight Infiniband DDR ports, each with a theoretical unidirectional bandwidth of 16 Gbp/s. The entire Intrepid system consists of 128 dual-core file servers, having 8 GB memory each. These file servers, as earlier noted, connect to the 10-Gigabit Ethernet network through their 10-Gigabit Ethernet ports. The I/O nodes connect to this 10-Gigabit Ethernet network as well. We note that the peak unidirectional bandwidth of each 10-Gigabit Ethernet port of the I/O node is *limited to 6.8 Gb/s* by the internal collective network that feeds it (IBM Blue Gene Team, 2008). From a theoretical performance standpoint, however, the network links connecting the file servers to the 10-Gigabit Ethernet network will become a bottleneck.

3.2 Overview of the IBM BG/P Checkpointing Library

IBM provides a special user-level *full checkpointing* library (Sosa and Knudson, 2007) for BG/P applications. This library provides support for *user-initiated checkpointing* where a user can insert checkpoint calls manually at critical areas in the application. Restarting an application is transparent and can be easily achieved by the user or system by setting certain environment variables (which point to the checkpoint file to be used for the restart) during application re-launch time. The checkpoint calls are made available to the user through a simple checkpointing API provided by IBM. Checkpoint-enabled applications are expected to call the *int BGCheckpointInit(char *path_to_checkpoint_directory)* routine at the beginning of the application. This routine initializes all the relevant data structures and is also used for the transparent restart of the applications when they are re-launched. The *int BGCheckpoint()* library call is used to take a snapshot/checkpoint of the program state at the instant at which it is called. All processes of the application should make this call to take a consistent global checkpoint. The user needs to ensure that no outstanding messages are present in the system when this checkpoint call is being made. It is also recommended that a Message Passing Interface (MPI) collective operation routine such as barrier synchronization (Snir and Otto, 1998) be called just before the *BGCheckpoint()*.

In addition to the above routines, IBM provides various other routines which allow applications to perform various tasks such as registering functions that can be called just before checkpointing (*int BGAtCheckpoint(args...)*), registering functions that can be called when continuing after a checkpoint (*int BGAtContinue(args...)*) or marking regions that can be excluded from the program state when the checkpoint is taken (*int BGCheckpointExcludeRegion(args...)*).

The IBM checkpointing library only provides support for full checkpointing. It provides no support for incremental checkpointing. Thus, for practical applicability, in this paper, we currently focus on studying applications, their memory usage, and checkpointing trends with respect to the full checkpointing method.

4 Overview of the Applications Used

In this section, we describe the applications that we run on the IBM Blue Gene machines. We study four popular applications from the fields of molecular dynamics and computational fluid dynamics.

4.1 Molecular Dynamics Simulations

In classical molecular dynamics (Car et al., 1985), a single potential energy surface is represented by a force field. However, most times, this level of representation is deemed inefficient. More accurate representations, involving atomic structure, electron/proton distribution, chemical reactions, and electronic behavior can be generated using quantum mechanical methods such as density functional theory. This area of molecular dynamics is known as Ab Initio (first principles) Molecular Dynamics (AIMD) (Marx and Hutter, 2000). GPAW (Grid Projected-Augmented Method) and CPMD (Carr-Parrinello Molecular Dynamics) are two very popular Ab Initio Molecular Dynamics codes. AIMD-based simulations tend to be highly complex with more stringent requirements for computational and memory resources as compared to traditional classical molecular dynamics code. The emergence of petascale machines like the IBM Blue Gene series has resulted in significant provisions for running such AIMD simulations efficiently and accurately. We thus choose two applications from this field, as described below, for our study of checkpointing trends on the IBM Blue Gene supercomputer.

4.1.1 Grid-Based Projector-Augmented Wave Application. The GPAW (Mortensen et al., 2005) application is a density functional theory (DFT) based code that is built on the projector-augmented wave (PAW) method and can use real-space uniform grids and multigrid methods. In the field of material and quantum physics, the Schrödinger equation (Cazenave,1989) is considered an important equation since it describes how the quantum state of a physical system changes in time. Various electronic structure methods (Foresman and Frisch, 1996) can be used to solve the Schrödinger equation for the electrons in a molecule or a solid, to evaluate the resulting total energies, forces, response functions, atomic structure, electron distributions, and other attributes of interest. The Projector Augmented Wave method is an electronic structure method for ab-initio molecular dynamics with full-wave functions. The term PAW is often also used used to describe the CP-PAW code developed originally by Blochl (Blochl et al., 2003). An advantage of PAW over other electronic structure methods is that PAW allows end-users to get rid of core electrons and work with soft pseudo-valence wave functions.

The GPAW application allows users to represent these pseudo-wave functions on uniform real-space orthorhombic grids. This makes it possible to run these modeling codes on very large systems, the results of which could be used to provide a theoretical framework for interpreting experimental results and even to accurately predict the material properties before experimental data actually becomes available. More information on GPAW, can be found on its website.

4.1.2 Carr-Parrinello Molecular Dynamics Application. The CPMD (Marx and Hutter, 2000; Andreoni and Curioni, 2000) code is another electronic structure method and a parallelized plane wave/pseudo-potential implementation of density functional theory, which targets *ab initio* quantum mechanical molecular dynamics plane wave basis sets. The CPMD code is based on the Kohn-Sham (Koch and Holthausen, 2001) DFT code, and it provides a rich set of features that have been successfully applied to calculate static and dynamic properties of many complex molecular systems such as water, proteins, and DNA bases, as well as various processes such as photoreactions, catalysis, and diffusion. Reactions and interactions in such systems are too complicated to be handled by classic molecular dynamics; but they can be successfully handled in the Carr-Parrinello method because they are calculated directly from the electron structure in every time step. CPMD's flexibility and high performance on many computer platforms have made it an optimal tool for the study of liquids, surfaces, crystals, and biomolecules.

CPMD runs on many computer architectures. Its well-parallelized nature, based on MPI, makes it a popular application that can take advantage of petascale systems, motivating us to choose it for this study.

4.2 Computational Fluid Dynamics Applications

Computational Fluid Dynamics (CFD) is a method of analyzing fluid and gas flow in structures and surfaces (Laval, 2008) using numerical methods and algorithms. CFD applications are used to predict how fluids flow and to predict the transfer of heat, mass, structural deformations, radiation, chemical reactions of the fluids and the solids with which they are in contact. Such CFD problems, which involve millions of calculations, place an immense demand on computing resources. The goal of a CFD application is to obtain a valid solution for a given CFD problem (constrained by certain boundaries) in a reasonable amount of time. With the availability of large scale machines like Blue Gene/P, it is possible to achieve a very high accuracy in computations involving CFD principles. CFD applications are used in various fields such as ocean modeling, thermal hydraulics, polymer processing, weather simulation, etc. In this paper, we study two popular CPD applications, namely the Nek5000 application and the Parallel Ocean Program (POP) application.

4.2.1 Nek5000. Nek5000 (Lottes and Fischer, 2004) is an open source, spectral element Computational Fluid Dynamics (CFD) code developed at the Mathematics and Computer Science Division at Argonne National Laboratory. The C and Fortran code, which won a Gordon Bell prize, focuses on the simulation of unsteady incompressible fluid flow, convective heat with species transport, and

magnetohydrodynamics. It can handle general two- and three-dimensional domains described by isoparametric quad or hex elements. In addition, it can be used to compute axisymmetric flows. Nek5000 is a time-stepping-based code and supports steady Stokes and steady heat conduction. It also features some of the first practical spectral element multigrid solvers, which are coupled to a highly scalable, parallel, coarse-grid solver.

The Nek5000 application was chosen for this study because it is highly scalable and can scale to processor counts of over 100,000, typical of petascale computing platforms. In addition, the Nek5000 software is used by many research institutions worldwide with a broad range of applications, including ocean current modeling, combustion, spatiotemporal chaos, the interaction of particles with wall-bounded turbulence, thermal hydraulics of reactor cores, and transition in vascular flows.

4.2.2 The Parallel Ocean Program. Parallel Ocean Program (POP), developed at the Los Alamos National Laboratory, is a three-dimensional ocean circulation model, designed to study an ocean's climate system. The POP application is an integral part of the SPEC 2007 Benchmark suite. The POP application has been used to perform the highest resolution global ocean simulation at Los Alamos National Laboratory. POP is used in a variety of applications, including very high resolution eddy-resolving simulations of the ocean and as the ocean component of coupled climate models like the Community Climate System Model.

POP is a descendant of the Bryan-Cox model, that has been used frequently for ocean climate simulations. POP has substantial improvements (Jones et al.,2005) over the earlier model. The POP application solves the three-dimensional equations for fluid motions on the sphere under hydrostatic and Boussinesq approximation. The spatial derivatives are computed using finite-difference discretizations, which are formulated to handle any generalized orthogonal grid on a sphere. These grids include the dipole and tripole grids which shift the North Pole singularity into land masses to avoid time step constraints due to grid convergence. Time integration of the model is split into two parts. The three-dimensional vertically-varying (baroclinic) tendencies are integrated explicitly using a leapfrog scheme. The very fast vertically-uniform (barotropic) modes are integrated using an implicit free surface formulation in which a preconditioned conjugate gradient solver is used to solve for the two-dimensional surface pressure. A wide variety of physical parameterizations and other features are also available in the model.

Portability in POP is achieved by isolating all communication routines into a small set of modules which can be modified for specific architectures (Jones et al., 2005). POP has been validated on many platforms including the IBM Blue Gene/L (Kerbyson et al., 2005). The combination of fine-granularity simulations, provided in POP, to resolve ocean eddies (which can impact dynamics of the ocean significantly) and long time scales required for climate and deep ocean circulation necessitates many computational cycles; making POP a good candidate for our study on BG/P.

5 Theory and Experiments

In this section we describe the experimental methodology used to conduct this study. We also describe our checkpointing model and use it in conjunction with the observed application memory trends to gain insight into optimal checkpointing parameters.

The scope of our study measures and analyzes the following trends:

1. How the application memory usage varies over application execution time, and
2. How the application memory usage varies with system size.

These memory trends were observed for the GPAW, CPMD, Nek5000, and the POP applications for varying system sizes up to 65,536 cores. These trends were then analyzed to determine the checkpoint frequency and checkpoint duration using the optimum checkpoint model. In the next few subsections we discuss the mechanism used to measure the memory usage and the details of our checkpointing model.

5.1 Recording Memory Usage Patterns

In the "full memory checkpointing" technique, a snapshot of the entire process memory for every process is taken. In our experiments, we run the various application codes and record the text memory, data memory, stack memory, and process-related information. The text memory contains the instruction-specific code. The data memory (which includes the heap memory) along with the stack memory, manages the local, global, static, and declared variables, as well as memory requested by various system (new, malloc, calloc) calls. Note that applications running on the IBM Blue Gene Compute Node Kernel (CNK) are statically linked, resulting in the text memory portion remaining constant during the lifetime of the application.

In our setup, a minimal amount of instrumentation code was inserted at the startup of each application that could record the memory usage for a process. An optimal method to achieve this, transparently to the application, would have been through *constructor* methods made available in a statically linked external library. However, the BG/P compilers, provided by IBM, do not provide this option. As an alternative, we invoke this measuring code soon after the entry point of the application.

From an implementation perspective, the memory-usage measuring code is based on a timer function that sets up a timer interrupt for a certain defined interval. As shown in Figure 3, the interrupt handler method when invoked measures the amount of memory used and records it. The *getrusage()* function, available in POSIX.1-2001, provides the measurements of the resources (indicated in Table 1) used by the current process. In our case, we use this routine in

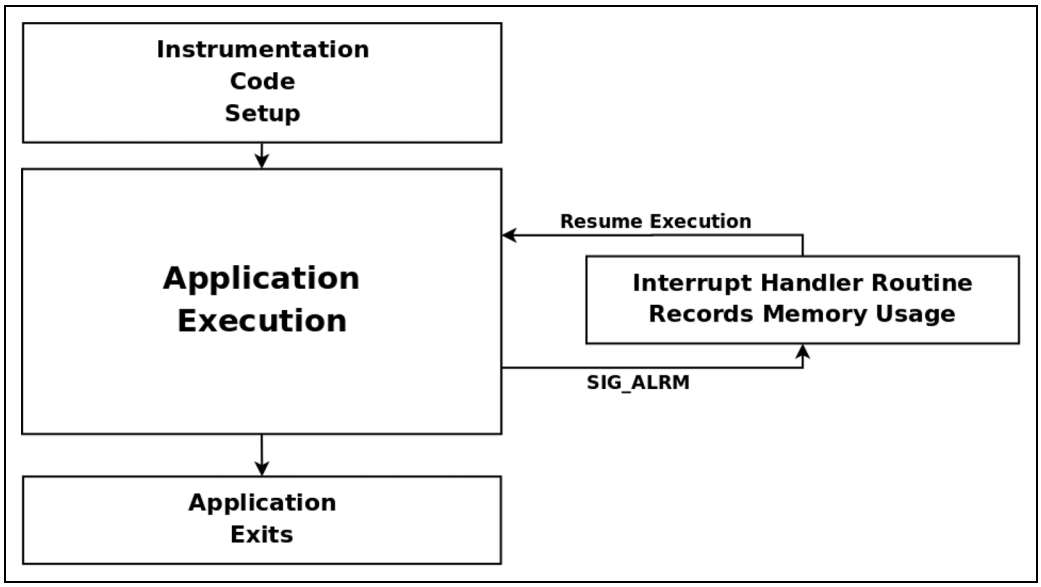

Figure 3. Instrumentation code.

our instrumentation code, to measure the resident memory size (indicated by the *ru_maxrss* field in Table 1) for the process. The resident set size of a process refers to the amount of physical memory the process is using. Since the CNK operating system does not support virtual memory, the *getrusage()* routine takes into consideration only the currently active pages of the process.

5.2 The Optimum Checkpointing Model

Runtime on large systems like Intrepid is a valuable commodity, which users wish to use in the most optimal manner. While checkpointing is a necessary activity, users wish to devote only a certain percentage of application execution time to this procedure. Knowledge about checkpoint duration and frequency, based on such constraints, would help the user make more informed decisions and tradeoffs between their resource usage and application resiliency. We attempt to provide this information through the "optimum checkpoint model" discussed in this section. Our checkpoint model has been largely influenced by past research on other checkpointing models (Young, 1974; Daly, 2006) in this field.

To arrive at the analytical model of the optimum checkpointing scheme on the Blue Gene/P system, let us consider an example application as shown in Figure 4. Like a majority of scientific applications, this application has a constant memory pattern for a majority of its execution period. The figure is a diagrammatic representation of the application with the checkpointing feature enabled.

Here T is the total execution time, including the time required to perform all the checkpoints; T_s is the time required to complete one full checkpoint operation; N is the optimum number of checkpoints to be performed during the length of the application execution; and t is the optimum time interval between two checkpoints when the application resumes execution.

Table 1. The *rusage* Structure

Type	Field	Function
Struct timeval	ru_utime;	user time used
Struct timeval	ru_stime;	system time used
Long	ru_maxrss;	maximum resident set size
Long	ru_ixrss;	integral shared memory size
Long	ru_idrss;	integral unshared data size
Long	ru_isrss;	integral unshared stack size
Long	ru_minflt;	page reclaims
Long	ru_majflt;	page faults
Long	ru_nswap;	Swaps
Long	ru_inblock;	block input operations
Long	ru_oublock;	block output operations
Long	ru_msgsnd;	messages sent
Long	ru_msgrcv;	messages received
Long	ru_nsignals;	signals received
Long	ru_nvcsw;	voluntary context switches
Long	ru_nivcsw;	involuntary context switches

Let n be the number of cores the application is run on, M be the mean memory usage per core, and B be the unidirectional bandwidth from all the compute nodes to storage disks that is available to the entire application. Based on these parameters, the time T_S required to save the state of all processes (cores) during a single checkpoint can be given by:

$$T_s = \frac{M \times n}{B}. \quad (1)$$

If T_A is the actual length of time the application needs to complete without checkpointing enabled, the total application runtime with checkpointing enabled, i.e. T, is given by

$$T = T_A + (N \times T_s). \quad (2)$$

Note that, for long running applications on large systems, end-users tend to have information on the total execution times (i.e. T_A) based on historical data or past runs performed. Frequently, users wish to devote only a certain

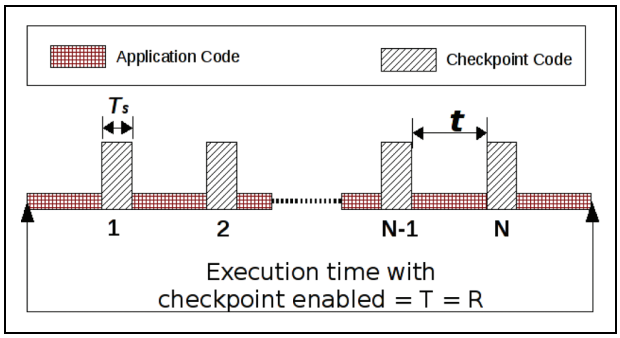

Figure 4. Application execution with checkpoint enabled.

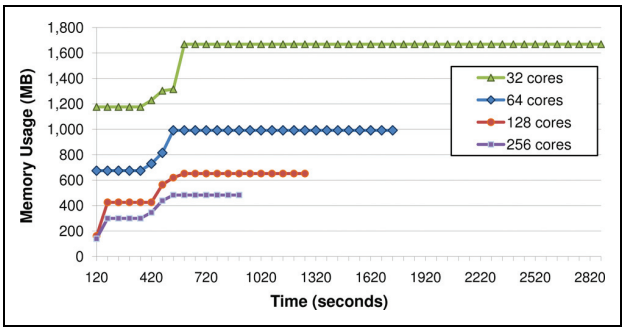

Figure 5. GPAW memory consumption for small systems.

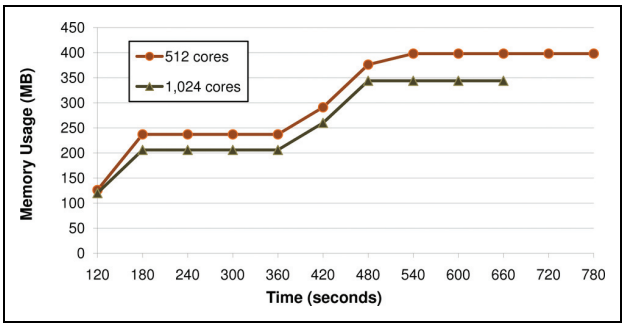

Figure 6. GPMD memory consumption for large systems.

percentage, X, of the application execution time, T_A, for checkpointing (note that in such situations, users increase the reservation time by $X\%$ as well). Thus, the overall execution time of an application with checkpoints enabled can also be described as:

$$T = T_A + (X \times T_A). \tag{3}$$

Based on equations (1), (2), and (3), one can estimate the total number of checkpoints, N, possible as follows:

$$T_A + (N \times T_s) = T_A + (X \times T_A) \tag{4}$$

$$N \times T_s = X \times T_A. \tag{5}$$

Therefore, N can be computed as follows:

$$\therefore N = \left\lfloor \frac{XT_AB}{Mn} \right\rfloor. \tag{6}$$

We can, thus, derive the number of optimum checkpoints based on: (a) the percentage of the application execution time dedicated for checkpointing, (b) the bandwidth from the compute nodes to the file servers, and (c) the total amount of data to be checkpointed. Having this information provides users the flexibility to choose areas they wish to checkpoint. In practice, it may be difficult to predict an accurate value for B since several applications may use the I/O and network resources at the same time. However, the user can make educated guesses based on their system size and file system architecture, as we show in the next section. (On a side note, one might speculate that since bandwidth is an important factor in determining the number of checkpoints, it might be worthwhile spreading the problem among more nodes [for example, by running a 16K process job on 16K nodes; instead of 4K nodes with four cores each] to increase the I/O bandwidth and reduce the memory usage per core. In practice, this turns out to be an expensive idea since end-users will typically get charged for cumulative processing cycles [i.e. a user would be charged for time on 16K nodes $\times$ 4 cores each $=$ 64K cores; as compared with just 16K cores] for such jobs.)

Finally, the frequency of checkpointing or the time interval between two checkpoints can be calculated as follows:

$$t = \frac{T_A}{N}. \tag{7}$$

Simplifying the above equation with substitutions from equation (6)

$$t = \frac{Mn}{XB}. \tag{8}$$

Having checkpoint interval information makes it easier to develop independent libraries that can be compiled into the application and can used timer-based methods to checkpoint periodically.

5.3 Application Evaluation

In this section, we carry out the evaluation of the GPAW, CPMD, Nek5000, and the POP applications on the Intrepid system.

All the above applications are strong-scaled problems. Strong scaling is a methodology used where the overall problem size is kept constant but the number of processors that the application is executing on is varied. (For weak scaling, the problem size *per processor* is kept constant and as the number of processors is increased, the problem size is increased as well.)

Applications on the Intrepid system can be run in three modes: (a) SMP mode: In this mode, only one process, with a maximum of four threads can be launched on each compute node with each thread using a core. (b) Dual mode: In this mode, two processes with a maximum of two threads each can be launched with each thread using a core. The 2 GB is equally divided between the two processes. (c) Virtual mode: In this mode, a single process per core (i.e. four processes per compute node) can be launched. The 2 GB memory is equally divided among all the four processes.

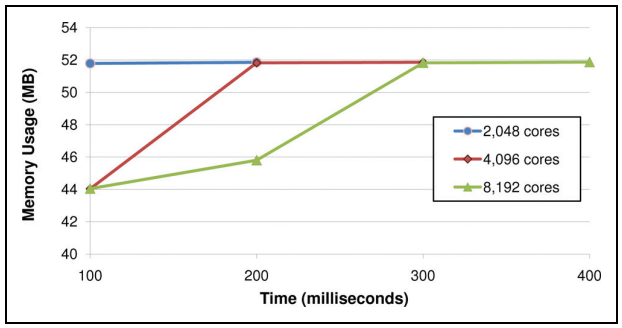

Figure 7. CPMD memory consumption.

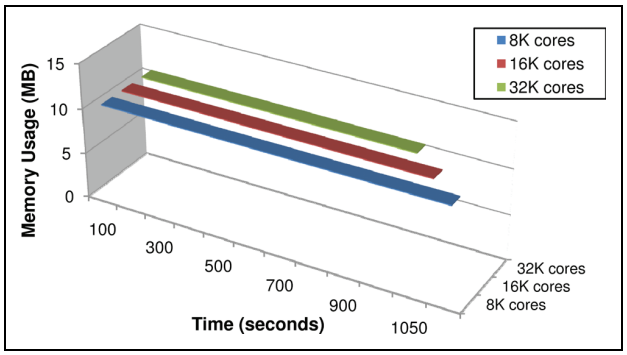

Figure 8. Nek5000 memory consumption.

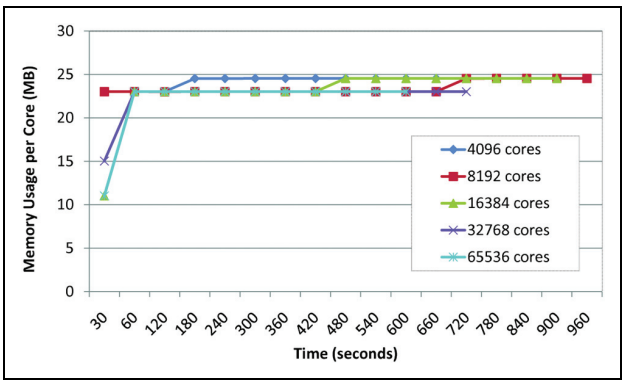

Figure 9. POP memory consumption.

The GPAW application (version 0.4) being evaluated, consists of 256 water molecules with 2,048 electrons, 1,056 bands, and 112^3 grid points, with a grid spacing of 0.18. This application was run on the Intrepid system, with the compute node count varying from 32 cores to 1,024 cores. The problem size was kept constant between all runs of the application. Note that memory in GPAW is dynamically allocated. The application was run in SMP mode with a single thread. Figure 5 shows the memory trends for up to 256 cores and Figure 6 shows it for 512 and 1,024 cores. The x-axes show the total application execution time and the y-axes show the memory usage per core. As can be seen from the graphs, the memory requirements for the GPAW application grow relatively slowly against the total time execution as the system size increases. We also observe that the application execution time decreases with increasing system size. The graph for GPAW clearly shows the decrease in memory footprint per core as the number of processors (system size) increases. The memory usage per core remains constant once the peak is attained, indicating a constant possible checkpointing time in later stages of the code.

Figure 7 shows memory trends for the CPMD application. The CPMD application was run in SMP mode with four threads on a system size of 8,192 cores. While the CPMD memory consumption trend is similar to the GPAW application, we notice that CPMD memory consumption increases a little slowly as system size increases. The important difference is that the memory consumption for the majority of the execution time remains the same irrespective of the system size.

Figure 8 shows the memory consumption trend for Nek5000. The Nek5000 application was run in virtual mode on a system size ranging from 8,192 cores to 32,768 cores. The 3–D graph in Figure 8 shows the system size on the z-axis and the application execution time and memory usage (for the various system sizes), on the x-axis and y-axis, respectively. In Nek5000, memory is allocated as soon as the application starts up and remains the same for the execution lifetime of the process, irrespective of the number of processors the application is finally meant to run on or the size of the problem set assigned to a particular node. Like CPMD, Nek5000 memory consumption does not vary with a change in system size.

Figure 9 shows memory consumption for the POP ocean modeling application. The POP application was run in virtual mode for system sizes ranging from 4,096 cores to 65,536 cores. The trends seen for the POP application show that memory usage increases rapidly once the application starts up and it remains constant for the lifetime of the application.

5.4 Computing Optimum Checkpointing Values

We briefly discussed the I/O and the network infrastructure of the Intrepid system in Section 3. As discussed, the bandwidth between the 10-Gigabit Ethernet network and the file servers is theoretically 2 Tbps. When an application uses the entire BG/P system, all the 640 I/O nodes can theoretically deliver up to a maximum of 4.25 Tbps. This indicates that the bandwidth bottleneck for the maximum data throughput lies more towards the file servers than the I/O nodes when the system is running at full capacity. However, since our study involves only using up to 16 racks (65,536 cores) out of the 40 racks (163,840 cores) available, we only use a maximum of 256 I/O nodes, thus limiting the I/O node bandwidth to 1,740 Gbps (i.e. 256×6.8 Gbps).

Moreover, as we are utilizing only a fraction of the total system capacity, the available file server I/O bandwidth for our runs is also limited by the other applications that are performing I/O bound operations. Each I/O node is equipped with a network interface capable of delivering 6.8 Gb/s. We based our optimum checkpoint value calculation on two cases. In the first case (B_{30}), we assume that we

Table 2. Computed Values for GPAW

				30% Bandwidth			60% Bandwidth		
n	M	T_A	X	B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
32	1650	3168	0.3	127.5	2	1584	255	4	792
64	1000	1914	0.3	255	2	957	510	4	478.5
128	650	1386	0.3	510	2	693	1020	5	277.2
256	475	990	0.3	1020	2	495	2040	4	247.5
512	400	858	0.3	2040	2	429	4080	5	171.6
1024	350	726	0.3	4080	2	363	8160	4	181.5

Table 3. Computed Values for CPMD

				30% Bandwidth			60% Bandwidth		
n	M	T_A	X	B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
2048	51.82	220	0.4	2040	1	220	4080	3	73.33
4096	51.85	330	0.4	4080	2	165	8160	5	66
8192	51.87	440	0.4	8160	3	146.67	16320	6	73.33

Table 4. Computed Values for Nek5000

				30% Bandwidth			60% Bandwidth		
n	M	T_A	X	B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
8192	10.52	1080	0.05	8160	5	216	16320	10	108
16384	10.52	1050	0.05	16320	4	262.5	32640	9	116.67
32768	10.52	1000	0.05	32640	4	250	65280	9	111.11

Table 5. Computed Values for POP

				30% Bandwidth			60% Bandwidth		
n	M	T_A	X	B_{30}	N_{30}	t_{30}	B_{60}	N_{60}	t_{60}
4096	25.13	960	0.07	4080	2	480	8160	5	192
8192	25.13	960	0.07	8160	2	480	16320	5	192
16384	25.13	900	0.07	16320	2	450	32640	4	225
32768	23.57	900	0.07	32640	2	450	65280	5	180
65536	23.57	900	0.07	65280	2	450	130560	5	180

are able to makes use of 30% of the total bandwidth provided by each I/O node interface. In the second case (B_{60}), we optimistically assume that we make use of 60% of the total bandwidth provided by each I/O node interface.

On the Intrepid system, the I/O node to compute node ratio is 1:64. Each compute node consists of a quad-core processor. When an application is run in the virtual node mode or SMP mode with four threads, the 64 compute nodes provide a total of 256 cores. The Nek5000 and POP applications were run in virtual node mode and the CPMD application was run in SMP mode with four threads.

Assuming, $B_{I/O}$ to be the total bandwidth available per I/O node and n to be the number of cores the application is running on, the total bandwidth available to the Nek5000 application, the CPMD application, and the POP application can be computed by:

$$B = \left(\frac{n}{64 \times 4}\right) \times B_{I/O}. \quad (9)$$

The GPAW application is executed in SMP mode with one thread, with only core being used on each compute node. The total bandwidth available to the GPAW application can be computed by:

$$B = \left(\frac{n}{64}\right) \times B_{I/O}. \quad (10)$$

For case 1 with 30% of the total bandwidth available, we have $B_{I/O} = 0.30 \times 6.8Gb/s = \frac{0.30 \times 6800}{8} = 255$ MB/s.

For case 2 with 60% of the total bandwidth available, we have $B_{I/O} = 0.60 \times 6.8Gb/s = \frac{0.60 \times 6800}{8} = 510$ MB/s.

Based on the the equations from Section 5.2 and the observations in Section 5.3, we can compute the optimum

checkpointing parameter values for each of these applications.

Let us assume that the user is willing to dedicate 3% to 7% of the application execution time for performing checkpoint operations. So we have X varying from 0.03 to 0.07 for the different applications.

Based on equations 6, 8, 9 and 10 and the graphs obtained in the previous section, we compute the optimum checkpointing parameters for the applications and tabulate them. Tables 2, 3, 4, and 5 show the calculated checkpointing values for the four applications where n is the number of cores, M is the average memory usage per core in megabytes, T_A is the application execution time in seconds (milliseconds for the CPMD application), B is the unidirectional bandwidth from all the compute nodes to storage disks in MB/s, N is the number of optimal checkpoints, and t is the checkpoint interval in seconds (milliseconds for the CPMD applications).

Table 2 shows the various checkpoint values for node counts ranging from 32 to 1,024 nodes and X set to 3%. We see that for B_{30}, the number of optimal checkpoints is 2 for all node ranges and for B_{60}, the optimal checkpoint count lies between 4 and 5. Thus, we see that the checkpoint frequency will heavily depend on the available I/O bandwidth, which can be challenging to determine.

The remaining tables show similar calculations for the other applications.

6 Conclusions and Future Work

In this paper, we discussed checkpointing trends for applications running on leadership class machines such as the IBM Blue Gene/P Intrepid system at Argonne National Laboratory. Ranked #5 on the Top500 November 2008 ranking, the Intrepid has 163,840 processors and a peak performance of 557 teraflops. We also presented an analytical model for efficiently computing the optimum checkpoint frequencies and intervals and studied four applications: the Grid-Based Projector-Augmented Wave application (GPAW), the Carr-Parrinello Molecular Dynamics application (CPMD), the Nek5000 computational fluid dynamics application, and Parallel Ocean Program (POP). We also showed with the help of experimental data and computed values how application scaling characteristics influence checkpoint-related decisions.

Our current work considered "full checkpointing," where the entire program state of the processes is stored during the checkpoint operation. We chose this approach because the IBM checkpointing library currently supports only full checkpointing. We are conducting a similar study for incremental checkpointing of applications on IBM BG/P, which will be useful for incremental checkpointing libraries built in the future for this machine. Our current study has been conducted on up to 65,536 processors of the Intrepid system. Further research in this direction will be to profile the content of the application memory in order to place checkpoints at critical junctures of the code.

Note

1. This gure was taken from the "IBM System Blue Gene/P Solution: Blue Gene/P Application Development" redbook (Sosa and Knudson, 2007).

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research also used resources of the Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

Andreoni, W. and Curioni, A. (2000). New Advances in Chemistry and Materials Science with CPMD and Parallel Computing. *Parallel Computing* **26**(7-8): 819–842.

Barigazzi, G. and Strigini, L. (1983). Application-Transparent Setting of Recovery Points. In *FTCS*.

Blochl, P., Forst, C. and Schimpl, J. (2003). Projector Augmented Wave Method: Ab Initio Molecular Dynamics with Full Wave Functions. 26-1.

Car, R., Parrinello, M., Schmidt, J., Sebastiai, D. et al. (1985). Unified Approach for Molecular Dynamics and Density-Functional Theory.

Cazenave, T. (1989). An Introduction to Nonlinear Schrodinger Equations.

Daly, J. T. (2006). A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems* **22**(3): 303–312.

Desai, N., Bradshaw, R., Lueninghoener, C., Cherry, A., Coghlan, S. and Scullin, W. (2008). Petascale System Management Experiences. In *LISA*.

Dongarra, J., Meuer, H. W. and Strohmaier, E. Technical report.

Dongsheng, W., Weimin, Z., Dingxing, W. and Meiming, S. Checkpointing and Rollback Recovery for Network of workstations. *Science in China Series E: Technological Sciences* **42**: 207–214.

Foresman, J. and Frisch, E. (1996). Exploring Chemistry with Electronic Methods.

GPAW Website: https://wiki.fysik.dtu.dk/gpaw

Guohong, C. and Singhal, M. (1998). On Coordinated Checkpointing in Distributed systems. In *TPDS*, vol. 9-12, pp. 1213–1225.

IBM Blue Gene Team. (2008). Overview of the IBM Blue/Gene Project. 52-1/2.

Jones, P. W., Worley, P. H., Yoshida, Y., White III, J. B. and Levesque, J. (2005). Practical performance portability in the Parallel Ocean Program (POP). *Concurrency and Computation: Practice and Experience* **17**(10): 1317–1327.

Kerbyson, Darren, J. and Jones, P. W. (2005). A Performance Model of the Parallel Ocean Program. *International Journal of High Performance Computing Applications* **19**(3): 261–276.

Koch, W. and Holthausen, M. (2001). A Chemist's Guide to Density Functional Theory.

Koo, R. and Toueg, S. (1986). Checkpointing and Rollback-Recovery For Distributed Systems. In *ACM Fall Joint Computer Conference*.

Laval, B. (2008). Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics.

Li, C. and Fuchs, W. (1990). CATCH - Compiler Assisted Techniques for Checkpointing. In *FTCS*.

Liu, Y., Nassar, R., Leangsuksun, C., Paun, M. and Scott, S. L. (2008). An Optimal Checkpoint/Restart Model for Large Scale High Performance Computing System, pp. 1–9.

Lottes, J. and Fischer, P. (2004). Hybrid multigrid/Schwarz algorithms for the spectral element method. *Journal of Scientific Computing* 45–78.

Marx, D. and Hutter, J. (2000). Ab-initio Molecular Dynamics: Theory and Implementation. NIC series, Vol. 3.

Mortensen, J. J., Hansen, L. B. and Jacobsen, K. W. (2005). Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B* **71**(3): 035109, Jan.

Plank, J. (1997). An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372. University of Tennesse, Knoxville. Available at http://www.cs.utk.edu/plank/plank/papers/CS-97-372.html.

Plank, J., Beck, M., Kingsley, G. and Li, K. (1995). Libckpt: transparent checkpointing under Unix. In Usenix Winter Technical Conference, pp. 213–223.

The Parallel Ocean Program (POP) Overview, website: http://climate.lanl.gov/Models/POP

Sancho, J., Petrini, F., Johnson, G. and Frachtenberg, E. (2004). On the Feasibility of Incremental Checkpointing for Scientific Computing. In *IPDPS*.

Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K. and Stodghill, P. (2004). Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, p. 38, Washington, DC, USA. IEEE Computer Society.

Silva, L. and Silva, J. (1998). System-Level versus User-Defined Checkpointing. In *SRDS*.

Snir, M. and Otto, S. (1998). MPI—The Complete Reference: The MPI Core.

Sosa, C. and Knudson, B. (2007). IBM System Blue Gene/P Solution: Blue Gene/P Application Development. Available at http://www.redbooks.ibm.com/abstracts/sg247287.html

Young, J. W. (1974). A first order approximation to the optimum checkpoint interval. *Communications of the ACM* **17**(9): 530–531.

Zomaya, A. (1995). Parallel and Distributed Computing Handbook.

Author's Biographies

Rinku Gupta is a senior scientific developer at Argonne National Laboratory. In the past 10 years, she has worked on several aspects of system and infrastructure development for enterprise high-performance computing. She received her MS degree in computer science from Ohio State University in 2002. Her research interests primarily lie towards middleware libraries, programming models, and fault tolerance in high-end computing systems. She is currently involved in researching coordinated fault tolerance techniques for system software on current petascale machines and emerging exascale supercomputers.

Harish Naik currently works as a pre-doctoral researcher at the Argonne National Laboratory's Mathematics and Computer Science division. He graduated from the University of Illinois at Chicago with a master's degree in computer science in the fall of 2008. He finished his bachelor's degree in computer science and engineering from R. V. College of Engineering, Bangalore in the year 2004. His research interests are broadly in the area of system software for large scale machines and parallel programming models. Particularly his interests lie in the areas of operating systems with respect to design and implementation, performance measurement, and noise evaluation. He is also involved in the design and implementation of policy definition and execution mechanisms for fault-tolerance and recovery enabled resource managers and software components.

Pete Beckman is a recognized global expert in high-end computing systems. During the past 20 years, he has designed and built software and architectures for large-scale parallel and distributed computing systems. After receiving his Ph.D. degree in computer science from Indiana University, he helped found the university's Extreme Computing Laboratory, which focused on parallel languages, portable run-time systems, and collaboration technology. In 1997 he joined the Advanced Computing Laboratory at Los Alamos National Laboratory, where he founded the ACL's Linux cluster team and launched the Extreme Linux series of workshops and activities that helped catalyze the high-performance Linux computing cluster community.

He also has been a leader within industry. In 2000 he founded a Turbolinux-sponsored research laboratory in Santa Fe that developed the world's first dynamic provisioning system for cloud computing and HPC clusters. The following year, he became vice president of Turbolinux's worldwide engineering efforts, managing development offices in the US, Japan, China, Korea, and Slovenia.

He joined Argonne National Laboratory in 2002, as Director of Engineering, and later as Chief Architect for the TeraGrid, he designed and deployed the world's most powerful grid computing system for linking production HPC computing centers for the National Science Foundation. After the TeraGrid became fully operational, he started a research team focusing on petascale high-performance software systems, Linux, and the SPRUCE system to provide urgent computing for critical, time-sensitive decision support.

In 2008 he became the director for the Argonne Leadership Computing Facility, which is home to one of the world's fastest open science supercomputers in production. He also leads Argonne's exascale computing strategic initiative and explores system software and programming models for exascale computing.

To be removed before publishing: